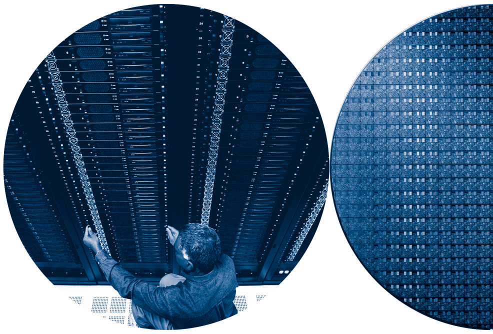
Intelligent RAID 6 Theory Overview and Implementation

www.intel.com/design/
storage/intelligent_raid.htm



Abstract

RAID 5 systems are commonly deployed for data protection in most business environments. However, RAID 5 systems only tolerate a single drive failure, and the probability of encountering latent defects of drives approaches 100 percent as disk capacity and array width increase. To address this issue, RAID 6 systems will soon be widely adopted because RAID 6 systems protect critical user data even when two drives or more fail concurrently. The Mean Time to Data Loss (MTDL) of RAID 6 systems is greater than RAID 5 systems; therefore, RAID 6 systems provide much better data protection than RAID 5. The RAID 6 theory is based on Maximum Distance Separable (MDS) coding, as well as Galois Field (GF) mathematics, which is introduced briefly in this paper. In addition, the P and Q parity computation and data recovery schemes are presented to help readers transition to RAID 6 systems rapidly. Lastly, the RAID 6 acceleration features of the Intel® IOP333 I/O processor are discussed. The IOP333 processor offloads the host CPU from intensive RAID 6 computations performed on a byte-by-byte basis in the data-path, allowing implementation of RAID 6 without compromising performance.

Introduction

The rising demand for capacity, speed and reliability of storage systems has expedited the acceptance and deployment of Redundant Array of Independent Disks (RAID) systems. RAID distributes data blocks among multiple storage devices to achieve high bandwidth input/output and uses one or more error-correcting drives for failure recovery. There are different RAID levels available for different data protection requirements. This paper focuses on RAID 6 systems and technology.

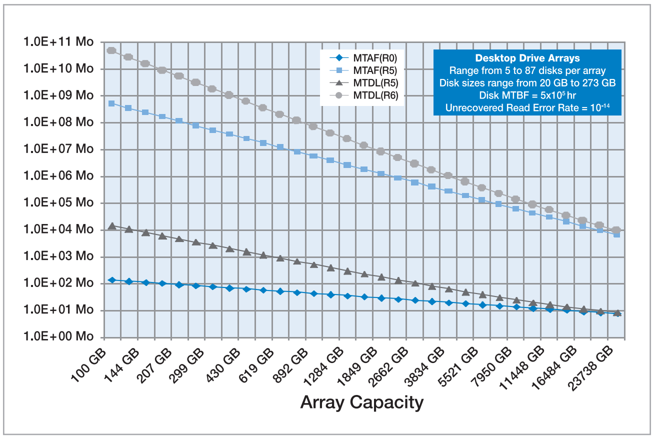
A RAID 5 system adds one parity drive in the form of parity elements in addition to the data; and is capable of recovering user data when one drive fails. When two drives fail, the user suffers data loss. A RAID 6 system, however, protects user data when two drives fail at the same time. In a practical scenario, the probability of two drives failing at the same time can be much higher than one might think. For example, as the capacity of drives used in a RAID 5 array grows, the RAID 5 rebuild time increases; during this time, the system is exposed to potential data loss. Furthermore, in a RAID 5 system, the operator occasionally pulls the wrong drive; thus, causing two drives to fail simultaneously. The ability of RAID 6 to tolerate simultaneous failures without loss of user data makes RAID 6 a better choice than RAID 5 for mission-critical applications. The remainder of the paper is structured as follows. The second section discusses the advantages of RAID 6 over RAID 5. The third section gives an overview of RAID 6 theory. The fourth section narrates a method for RAID 6 P and Q check value generation. The fifth section describes the recovery schemes when two drives fail at the same time. The sixth section discusses the programming of the Intel IOP333 I/O processor to facilitate RAID 6 algorithm implementation. The last section concludes this paper.

RAID 5 vs. RAID 6

In this section, the fail time and probability of failure of RAID 5 vs. RAID 6 systems will be examined. As mentioned before, the capacity of disks is growing each year to satisfy user demands. The effect of growing disk capacity is MTDL has decreased from approximately 2 years to 1 month for some large RAID 5 arrays. The MTDL for large RAID 5 arrays with high capacity disks is unacceptable in most scenarios. However for the same drive capacity (320 GB), and number of drives, the MTDL is approximately 100 years for a RAID 6 array. RAID 6 MTDL improves drastically over RAID 5 MTDL. Figure 1 compares the MTDL figures for RAID 0 (no redundancy), RAID 5 (tolerates 1 failure), and RAID 6 (tolerates 2 failures); accounting for the chance of encountering an unrecoverable read error during rebuild.

In Figure 1, the bottom line with diamonds shows the expected time before any one disk fails in the array. In an array with no redundancy (e.g. RAID 0), this would result in loss of data. The next line, with triangles, shows the MTDL for a RAID 5 array with the probability of finding a latent defect during rebuild factored in. Note that a RAID 5 array with greater than about 5 TB in total capacity could lose data multiple times in a single year. To illustrate the impact of latent defects in MTDL calculations, or Mean Time to Additional Failure (MTAF), the

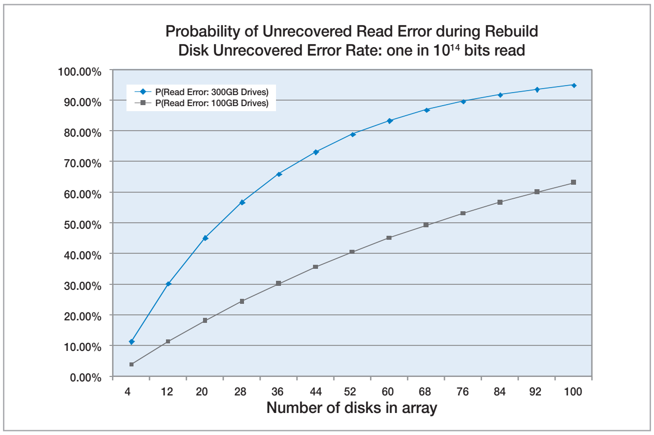
Figure 1 Time to Failure vs. Array Capacity



next curve, with squares, shows the probability of failure due to two disk failures, ignoring latent defects, for a RAID 5 array. Ignoring the impact of latent defects, the MTDL for a RAID 5 array remains quite good, even for arrays of 5 TB or greater capacity. The top line, with circles, shows the MTDL for a RAID 6 array taking into account the probability of finding latent defects. It shows the MTDL for RAID 6 arrays, even accounting for the impact of latent defects, is many orders of magnitude better than that for a comparable RAID 5 array.

To better understand how latent defects affect MTDL in RAID 5 arrays, we look at the probability of encountering a latent defect during a rebuild operation. If a RAID 5 controller encounters a defect during a rebuild operation, user data is lost because the failed disk and the defective sector represent two missing elements, exceeding the capability of RAID 5 to recover the lost data. Figure 2 shows the probability of finding a latent defect during array rebuild as array capacity grows. For extremely large arrays with high-capacity disks, it would be surprising not to encounter a latent defect during a rebuild operation. This graph assumes an error rate typical for desktop class drives. The probability is an order of magnitude lower for enterprise-class drives.

Figure 2 Latent Defects vs. Array Capacity



RAID 6 Theory Overview

A complete understanding of RAID 6 theory requires some knowledge of algebra and finite-field coding theory. The intent of this paper is not to bog down readers with minute details of algebra and coding theory. However, an overview of the theory is essential to implement the RAID 6 algorithm. Readers who are interested in a more in-depth discussion of coding theory are encouraged to read more detailed articles. ("Practical Error Correction Design for Engineers" by Neal Glover, Trent Dudley; Data Systems* Technology Corp; Rev 2nd edition — March 1, 1991).

RAID 6 technology uses MDS codes based on GF or Finite Field mathematics to encode data on the drives to protect data from error or erasure. The following paragraphs provide an overview of GF algebra and MDS encoding used in RAID 6 implementations.

MDS Codes

Irving Reed and Gus Solomon published a paper in 1960 that presented a new class of error-correcting codes referred to as Reed Solomon codes, which are an example of MDS codes. In this section we will discuss elements of Reed Solomon codes with characteristics similar to the MDS codes used in P and Q check value generation. Reed Solomon codes are non-binary systematic cyclic linear block codes. The encoding method generates a codeword consisting of a block of data symbols with a number of check symbols calculated from the block of data, which are appended at the end. The non-binary characteristic facilitates encoding of symbols consisting of several bits (normally 8 bits). The Reed Solomon encoding/decoding technique is widely used today in storage devices, wireless/mobile communications, satellite communications, digital television and high speed modems.

A Reed-Solomon code is specified as RS(n,k) with s-bit symbols (see Figure 3 below). n denotes the total number of symbols per codeword and k denotes the total number of data symbols per codeword. s represents the total of number of bits per symbol.

Figure 3 Reed-Solomon Codeword

<table>
<tr><td colspan="2">n (total symbols)</td></tr>
<tr><td>k (data symbols)</td><td>2t (parity symbols)</td></tr>
</table>

The number of parity symbols are $2t = n - k$. The RS code corrects up to t symbol errors and up to 2t erasures when the position of an error symbol is known. The MDS erasure coding used in P+Q RAID 6 is closely related to Reed-Solomon codes, where $2t = 2$ so the system corrects up to two erasures with two parity disks when the position of failure is known (as is always the case for disk failures or unrecovered read errors). For a symbol size s, the maximum codeword length is $n = 2^s - 1$. For example, when we use an eight bit symbol (i.e, GF(2^8), the maximum codeword length is $2^8 - 1 = 255$. A P+Q RAID 6 implementation forms a code word by taking a single byte from each of k data disks in the array and calculating two parity symbols (P and Q) stored on P and Q parity disks, and forming an n-symbol code word consisting of the k bytes read from k different data disks and the calculated P and Q parity values. Because the maximum code word length for GF(2^8) is 255, the maximum number of drives (n) that can be supported is 255 which includes data drives (253) and parity drives (2).

Galois Field (GF) Algebra Basics

A GF is a set of values that contains a finite number of elements. GF with 2^8 elements are denoted as GF(2^8) and have elements from integer 0 to $2^8 – 1$. Moreover, a finite field has an important property that arithmetic operations (add, subtract, multiply, divide) on field elements always have a result that is also in the field. The catch is the add, subtract, multiply and divide operations are not the same as the ones normally used in integer arithmetic, but are redefined to yield the desired properties in the finite field. The generation of elements in GF(2^8) will be investigated in the next paragraph.

Galois Field Element Generation

To enumerate elements in GF requires a primitive polynomial, which means the polynomial cannot be factored. There are several known primitive polynomials available based on the value of s (8). The GF elements are enumerated with the selected primitive polynomial, and the enumeration ends at the 2^s element. The following simple example demonstrates the generation of GF(2^2) elements. For example, let s = 2 (2^2 = 4 elements in GF) and let the primitive polynomial be $q(x) = x^2 + x + 1$. Enumeration begins with a primitive element in the field which is a root of q(x). If the value of x (say, α, called a primitive element) is a root of q(x), then it is true that $\alpha^2 + \alpha + 1 = 0$, or $\alpha^2 = \alpha + 1$. Addition is defined to be a bit-wise XOR of the coefficients of the polynomial, and multiplication corresponds to multiplication of polynomials of the primitive element, α. We start with {0, 1} elements then continue to enumerate subsequent field elements by multiplying the previous field element by x and taking the result modulo q(x) if it has degree >= 2. The third element is generated from $1 * \alpha = \alpha$. The fourth element is generated from $\alpha * \alpha = \alpha^2$. But because x is a root of the primitive polynomial, q(x), we know that $\alpha^2 = \alpha + 1$. Therefore, we end up with {0, 1, α, α+1} constituting the 4 elements in GF(2^2). In Figure 4, the table shows the GF elements expressed in various formats.

Figure 4 GF(2^2) Elements in Different Representations

Generated elements of GF(4)	Polynomial elements of GF(4)	Binary elements of GF(4)
0	0	00
a^0	1	01
a^1	a	10
a^2	a+1	11

Note that multiplying the last element in the table by x, yields,

$$\alpha*(\alpha +1) = \alpha^2+ \alpha = \alpha +1+ \alpha = 1,$$

which results in a field element (1) already defined. To continue this process simply cycle through the same four elements already calculated.

Galois Field Arithmetic Operations

To encode and decode MDS codewords, finite-field arithmetic is needed. Remember that the closed-field operation of finite-field arithmetic mandates that all arithmetic operations of elements in the field produce another element in the field. The following paragraphs discuss GF arithmetic operations and present a few examples.

Addition and subtraction of elements in GF fields are performed as an exclusive-OR. Because a GF is enumerated by taking powers of a primitive element, α; the i^{th} element is represented by α^i. Likewise, the j^{th} element is represented by α^j. Multiplication is achieved by addition of the exponents (i.e., $\alpha^i * \alpha^j = \alpha^{(i+j)}$,which is the $(i+j)^{th}$ element in the GF). In this case, addition is modulo 2^n-1, so for GF(2^8) if i+j >= 255, the result is the $(i+j – 255)^{th}$ element in GF(2^8). Note that since α^i is the i^{th} element in the set, then $i = \log_\alpha(\alpha^i)$. If the values of GF($2^8$) are enumerated as an indexed list, the index of the value is the log of the value. By swapping the columns and sorting in order of the values, α^i, one creates a log table to look up the log of any value. Multiplication in GF(2^8), then, is a simple matter of looking up the log values of the multiplicands, adding them (modulo 255), and looking up the inverse-log of the result. Software implementations will typically generate two tables (gflog and gfilog) to simplify these operations. Division in GF(2^8) is a similar operation with exponents subtracted rather than added (again, modulo 255). Examples of how to use gflog and gfilog tables to perform multiplication and division operations are presented in the next paragraph. The software to generate both tables is shown in Figure 5, and the primitive polynomial selected for s = 8 is $q(x) = x^8 + x^4 + x^3 + x^2 +1$. GF($2^8$) is constructed using a primitive element, $\alpha = 1$, which is a root of the polynomial, and therefore is a solution to the equation, $x^8 = x^4 + x^3 + x^2 +1$.

Figure 5 Logarithm Table Generation code

```
//*********************************************************************
// This routine is used to create logarithm and inverse loga-
   rithm tables for computing
// P&Q parity.  In our P&Q computation, we use all 1's for the
//coefficients of P.
//*********************************************************************
int gen_tables (int s)
{
  unsigned int                   b, index, gf_elements;
  //polynomial =  x^8 +  x^4 + x^3 + x^2 +1
  unsigned int                   prim_poly_8 = 0x11d;
  unsigned short                 *gflog, *gfilog;
  //convert s to the number of  elements for the table
  gf_elements = 1 << s;
  gflog  = (unsigned short *) malloc (sizeof (unsigned short) *
  gf_elements);
  gfilog = (unsigned short *) malloc (sizeof (unsigned short) *
  gf_elements);
  b = 1;
  for ( index = 0; index < gf_elements - 1; index++)
  {
    gflog[b] = (unsigned char) index;
    gfilog[index] = (unsigned char) b;
    b <<= 1;
    if ( b & gf_elements )
      b ^= prim_poly_8;
   }
   return 0;
}
```

In Figure 6 and 7, to simplify the multiplication and division examples, we only list the first 64 elements in hex value of the gflog and gfilog tables which are generated by the routine in Figure 5. These elements are used to demonstrate closed-field multiplication and division operations.

The value of x is represented in rows and columns so that the function result (in hex) is found by looking in the cell with row R and column C. For example, gflog(0x13) is found in the cell with row-1 and column 3 of the gflog table which gives a value of 0x0e. Likewise, gfilog(0x0e) is found by looking in the gfilog table in row 0, column e, which has the value 0x13, which is decimal 19.

Notation:

⊕   : denotes GF addition.
⊗   : denotes GF multiplication
÷   : denotes GF division
+   : denotes normal integer addition
0x64 : denotes hex value of 64
18   : denotes decimal value of 18

Multiplication:

2 ⊗ 8 = gfilog [gflog[2] + gflog[8]] = gfilog[1+3] = gfilog[4] = 0x10

0x12 ⊗ 5 = gfilog [gflog[0x12] + gflog[5]] = gfilog[0xe0+0x32] = gfilog[0x13] = 0x5a

Note: Addition here is normal integer addition performed modulo 255 (as opposed to the XOR function used for addition of two GF field elements).

Division:

0xd ÷ 0x11 = gfilog [gflog[0xd] - gflog[0x11]] = gfilog[0x68 - 0x64] = gfilog[4] = 0x10

2 ÷ 0xb = gfilog [gflog[2] + gflog[0xb]] = gfilog[0x1 - 0xee] = gfilog[0x12] = 0x2d

Figure 6 Logarithm Table -GF^{-1}(2^8)

gflog(x)	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	--	00	01	19	02	32	1a	c6	03	df	33	ee	1b	68	c7	4b
1	04	64	e0	0e	34	8d	ef	81	1c	c1	69	f8	c8	08	4c	71
2	05	8a	65	2f	e1	24	0f	21	35	93	8e	da	f0	12	82	45
3	1d	b5	c2	7d	6a	27	f9	b9	c9	9a	09	78	4d	e4	72	a6

Figure 7 Inverse Logarithm Table -GF(2^8)

gfilog(x)	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	1	2	4	8	10	20	40	80	1d	3a	74	e8	cd	87	13	26
1	4c	98	2d	5a	b4	75	ea	c9	8f	3	6	0c	18	30	60	c0
2	9d	27	4e	9c	25	4a	94	35	6a	d4	b5	77	ee	c1	9f	23
3	46	8c	5	0a	14	28	50	a0	5d	ba	69	d2	b9	6f	de	a1

The following basic properties of algebra apply to GF arithmetic operations. These properties are useful in manipulating GF elements.

- Commutative: $A \oplus B = B \oplus A;$
  $A \otimes B = B \otimes A;$
- Associative: $(A \oplus B) \oplus C = A \oplus (B \oplus C);$
  $(A \otimes B) \otimes C = A \otimes (B \otimes C);$
- Distributive: $(A \oplus B) \otimes C = A \otimes C \oplus B \otimes C;$
- Inverse: $A^{-1} = 1 \div A;$
- Multiplicative Inverse : For all x in GF, there exists y in GF such that $x \otimes y = 1$
- Multiplicative Identity : $1 \otimes A = A;$
- Additive inverse : For all **x** in GF, there exists **y** in GF such that $\mathbf{x} \oplus \mathbf{y} = \mathbf{0}$.
- Additive Identity : For all x in GF, $\mathbf{x} \oplus \mathbf{0} = \mathbf{x}$

P and Q Check Value Generation

In the previous paragraphs we have shown how to integrate gflog and gfilog tables and demonstrated using these tables how to simplify GF multiplication and division. Now we discuss how to generate P and Q check values when a stripe of data is being written or updated on to disks. If P and Q elements were always mapped to the same two disks, then every write operation serviced by the array would require updating of the same two disks. To avoid the bottleneck, P and Q check values are rotated through the disks in the array as follows. Figure 8 shows one possible layout of P and Q elements in a RAID 6 array. For stripe 0, drive 0 through 4 store data blocks but drives 5 and 6 store P and Q parity, respectively. For stripe 1, drives 1 through 3 and 6 store data blocks and drives 4 and 5 store P and Q parity respectively. The rotation continues until P and Q arrive back at their original positions, and the sequence repeats.

Figure 8 RAID 6 Data Blocks on Drives

Stripe	0	1	2	3	4	5	6
0	$D_{(0,0)}$	$D_{(0,1)}$	$D_{(0,2)}$	$D_{(0,3)}$	$D_{(0,4)}$	$P_{(0)}$	$Q_{(0)}$
1	$D_{(1,0)}$	$D_{(1,1)}$	$D_{(1,2)}$	$D_{(1,3)}$	$P_{(1)}$	$Q_{(1)}$	$D_{(1,4)}$
2	$D_{(2,0)}$	$D_{(2,1)}$	$D_{(2,2)}$	$P_{(2)}$	$Q_{(2)}$	$D_{(2,3)}$	$D_{(2,4)}$
3	$D_{(3,0)}$	$D_{(3,1)}$	$P_{(3)}$	$Q_{(3)}$	$D_{(3,2)}$	$D_{(3,3)}$	$D_{(3,4)}$
4	$D_{(4,0)}$	$P_{(4)}$	$Q_{(4)}$	$D_{(4,1)}$	$D_{(4,2)}$	$D_{(4,3)}$	$D_{(4,4)}$
5	$P_{(5)}$	$Q_{(5)}$	$D_{(5,0)}$	$D_{(5,1)}$	$D_{(5,2)}$	$D_{(5,3)}$	$D_{(5,4)}$
6	$Q_{(6)}$	$D_{(6,4)}$	$D_{(6,0)}$	$D_{(6,1)}$	$D_{(6,2)}$	$D_{(6,3)}$	$P_{(6)}$

P Check Value Generation

P check value generation is the same as RAID 5 parity computation. To simplify our examples, we assume each data block in a stripe contains only one byte of data. Therefore, $P_{(0)}$ check value for stripe 0 and $P_{(2)}$ check value for stripe 2 are computed as follows. The equation is simplified if we assume P is always one.

$$\begin{aligned} P_{(0)} &= P_0 \otimes D_{(0,0)} \oplus P_1 \otimes D_{(0,1)} \oplus P_2 \otimes D_{(0,2)} \oplus P_3 \otimes D_{(0,3)} \oplus P_4 \otimes D_{(0,4)} \\ &= D_{(0,0)} \oplus D_{(0,1)} \oplus D_{(0,2)} \oplus D_{(0,3)} \oplus D_{(0,4)} \end{aligned}$$

$$\begin{aligned} P_{(2)} &= P_0 \otimes D_{(2,0)} \oplus P_1 \otimes D_{(2,1)} \oplus P_2 \otimes D_{(2,2)} \oplus P_3 \otimes D_{(2,3)} \oplus P_4 \otimes D_{(2,4)} \\ &= D_{(2,0)} \oplus D_{(2,1)} \oplus D_{(2,2)} \oplus D_{(2,3)} \oplus D_{(2,4)} \end{aligned}$$

When a specific data block in a stripe is being updated, the P check value must be updated. This is normally referred to as a strip write. The following examples depict the computation of new P check values when strip 0 and strip 3 are written.

$$\begin{aligned} P_{(0)}\text{new} &= P_{(0)}\text{old} \oplus D_{(0,2)}\text{old} \oplus D_{(0,2)}\text{new} \\ P_{(3)}\text{new} &= P_{(3)}\text{old} \oplus D_{(3,1)}\text{old} \oplus D_{(3,1)}\text{new} \end{aligned}$$

Q Check Value Generation

To generate the Q check value we use the multiplier coefficients selected from GF elements in the gfilog table. The previous assumption that each data block contains only one byte of data applies here as well. The following examples show how $Q_{(0)}$ and $Q_{(5)}$ check values are generated when stripe 0 and stripe 5 are written.

$$\begin{aligned} Q_{(0)} &= g_0 \otimes D_{(0,0)} \oplus g_1 \otimes D_{(0,1)} \oplus g_2 \otimes D_{(0,2)} \oplus g_3 \otimes D_{(0,3)} \oplus g_4 \otimes D_{(0,4)} \\ &= 0x1 \otimes D_{(0,0)} \oplus 0x2 \otimes D_{(0,1)} \oplus 0x4 \otimes D_{(0,2)} \oplus 0x8 \otimes D_{(0,3)} \oplus \\ &\quad 0x10 \otimes D_{(0,4)} \end{aligned}$$

$$\begin{aligned} Q_{(5)} &= g_0 \otimes D_{(5,0)} \oplus g_1 \otimes D_{(5,1)} \oplus g_2 \otimes D_{(5,2)} \oplus g_3 \otimes D_{(5,3)} \oplus g_4 \otimes D_{(5,4)} \\ &= 0x1 \otimes D_{(5,0)} \oplus 0x2 \otimes D_{(5,1)} \oplus 0x4 \otimes D_{(5,2)} \oplus 0x8 \otimes D_{(5,3)} \oplus \\ &\quad 0x10 \otimes D_{(5,4)} \end{aligned}$$

When a specific data block in a stripe is being updated, the Q check value must also be updated. The following examples depict the computation of new Q check values when strip 2 or strip 3 are modified.

$$\begin{aligned} Q_{(2)\text{new}} &= Q_{(2)}\text{old} \oplus g_2 \otimes D_{(0,2)\text{old}} \oplus g_2 \otimes D_{(0,2)\text{new}} \\ &= Q_{(2)\text{old}} \oplus 0x4 \otimes D_{(0,2)\text{old}} \oplus 0x4 \otimes D_{(0,2)\text{new}} \\ &= Q_{(2)\text{old}} \oplus (0x4 \otimes (D_{(0,2)\text{old}} \oplus D_{(0,2)\text{new}})) \text{ - apply} \\ &\quad \text{distributive rule} \end{aligned}$$

$$\begin{aligned} Q_{(3)\text{new}} &= Q_{(3)\text{old}} \oplus g_3 \otimes D_{(3,1)\text{old}} \oplus g_3 \otimes D_{(3,1)\text{new}} \\ &= Q_{(3)\text{old}} \oplus 0x2 \otimes D^{(3,1)\text{old}} \oplus 0x2 \otimes D_{(3,1)\text{new}} \\ &= Q_{(3)\text{old}} \oplus (0x2 \otimes (D_{(3,1)\text{old}} \oplus D_{(3,1)\text{new}})) \text{ - apply} \\ &\quad \text{distributive rule} \end{aligned}$$

RAID 6 Recovery

RAID 6 recovers user data when two drives fail simultaneously. When two drives fail, to restore redundancy it is necessary to rebuild data that was on the failed drives to replacement drives using data from the remaining good drives. This paragraph discusses the rebuild process when two drives fail. There are four cases to consider: 1) P and Q drives fail, 2) P drive and one data drive fail, 3) Q drive and one data drive fail, 4) two data drives fail. The rebuild steps for each case are presented below. The recovery method for different stripes is different depending on whether the failed disks had P, Q, or user data as P and Q are rotated from one stripe to the next, but the recovery method for any given stripe is always one of the four cases listed above.

Recovery of P and Q Drive Failure

This is the easiest rebuild scenario. Since all data drives are intact, P and Q check values can be recomputed from data drives as mentioned in the previous paragraphs.

Recovery of Q and Single Data Drive Failure

The rebuild process is similar to RAID 5 recovery. Since P drive is intact, the data blocks can be rebuilt using P check value. This process is exactly the same as RAID 5 rebuild process. After the failed data blocks have been recovered, Q can be recomputed as mentioned above. The following example assumes drive 1 and the Q parity drive failed and demonstrates the recovery of the data block and Q check value of stripe 0.

$$D_{(0,1)} = P_{(0\,)} \oplus D_{(0,0)} \oplus D_{(0,2)} \oplus D_{(0,3)} \oplus D_{(0,4)}$$

$$Q_{(0)} = g_0 \otimes D_{(0,0)} \oplus g_1 \otimes D_{(0,1)} \oplus g_2 \otimes D_{(0,2)} \oplus g_3 \otimes D_{(0,3)} \oplus g_4 \otimes D_{(0,4)}$$

Recovery of P and Single Data Drive Failure

In this rebuild process, the Q check value is used to rebuild the data drive. Let’s assume drive 2 and the P drive failed. The first step in the rebuild process is to compute Q’ (Q computed based on the remaining data drives), then GF add Q’ to Q and GF multiply the result by g_2^{-1}. Drive 2 data blocks may be rebuilt through this process. After rebuilding drive 2 data blocks, the P check value can be recomputed using a normal XOR operation.

$$Q_0' = g_0 \otimes D_{(0,0)} \oplus g_1 \otimes D_{(0,1)} \oplus g_3 \otimes D_{(0,3)} \oplus g_4 \otimes D_{(0,4)}$$

$$D_{(0,2)} = g_2^{-1} \otimes (Q_{(0)} \oplus Q_0')$$

$$\begin{aligned} P_{(0)} &= P_0 \otimes D_{(0,0)} \oplus P_1 \otimes D_{(0,1)} \oplus P_2 \otimes D_{(0,2)} \oplus P_3 \otimes D_{(0,3)} \oplus P_4 \otimes D_{(0,4)} \\ &= D_{(0,0)} \oplus D_{(0,1)} \oplus D_{(0,2)} \oplus D_{(0,3)} \oplus D_{(0,4)} \end{aligned}$$

Recovery of Dual Data Drive Failure

The rebuild process to recover from a two data drive failure is the most complicated case. Let’s assume drive 1 and 2 failed; so, we have two equations and two unknowns.

$$D_{(0,1)} \oplus D_{(0,2)} = P_{(0)} \oplus D_{(0,0)} \oplus D_{(0,3)} \oplus D_{(0,4)}$$

$$g_1 \otimes D_{(0,1)} \oplus g_2 \otimes D_{(0,2)} = Q_{(0)} \oplus g_0 \otimes D_{(0,0)} \oplus g_3 \otimes D_{(0,3)} \oplus g_4 \otimes D_{(0,4)}$$

Using matrix inversion we solve for $D_{(0,1)}$ and $D_{(0,2)}$. $D_{(0,1)}$ and $D_{(0,2)}$ can be computed directly using the equations below. After we restore one of the data blocks, we can use the P check value to restore the other data block. This is an alternate way to compute the second data block.

$$D_{(0,1)} = (g_1 \oplus g_2)^{-1} \otimes ((g_2 \otimes (P_{(0)} \oplus P_{(0)}')) \oplus (Q_{(0)}) \oplus Q_{(0)}'))$$

$$D_{(0,2)} = (g_1 \oplus g_2)^{-1} \otimes ((g_1 \otimes (P_{(0)} \oplus P_{(0)}')) \oplus (Q_{(0)}) \oplus Q^{(0)\prime}))$$

$$P^{(0)\prime} = D_{(0,0)} \oplus D_{(0,3)} \oplus D_{(0,4)}$$

$$Q_{(0)}' = g_0 \otimes D_{(0,0)} \oplus g_3 \otimes D_{(0,3)} \oplus g_4 \otimes D_{(0,4)}$$

$$D_{(0,2)} = D_{(0,1)} \oplus (P_{(0)} \oplus P_{(0)}')$$

RAID 6 Acceleration with the Intel® IOP333

The Intel IOP333 is a multi-function device that integrates the Intel XScale® core (ARM* architecture compliant) with intelligent peripherals and dual PCI Express*-to-PCI-X bridges. The IOP333 is Intel's 6th generation I/O processor.

Intel® IOP333 Architecture Overview

The Intel IOP333 combines the Intel XScale core with powerful new features to create Intel's latest and most powerful intelligent I/O processor. This multi functional device is fully compliant with the PCI local bus Specification 2.3 and the PCI Express Specification, Revision 1.0a. The features included in the IOP333 are listed below:

- Intel XScale® Core @ up to 800 MHz
- PCI Express* 2.5 GHz x8 link
- Application Acceleration Unit (AAU) for XOR
- Application Accelerator for RAID 6, P + Q Implementation
- Address Translation Units (ATU)
- Dual-ported DDR 333 MHz/DDR2 400 MHz Memory Controller
- Peripheral Bus Interface (PBI)
- Two PCI Express-to-PCI-X* bridges to secondary PCI-X 133 MHz Bus Interfaces
- Two I²C Bus Interface Units
- Two DMA Controllers
- Messaging Unit (MU)

- 333 MHz Internal Bus
- Two UARTS
- Interrupt Controller and GPIOs

Intel® IOP333 Application Acceleration Unit (AAU)

The Application Acceleration Unit (AAU) provides low-latency, high throughput data transfer between the AAU and IOP333 local memory. It executes data transfers to and from the IOP333 local memory, checks for all zero results across local memory blocks, fills memory blocks, computes XOR values for 4, 8, 16 and 32 source data blocks and carries out GF multiplication. The main enhancement in the IOP333's AAU is the ability to accelerate the GF multiplication for RAID 6 implementations. The AAU operations can be chained together using descriptor entries. Chain descriptors provide efficient and effective programming interfaces to handle crossing scatter-gather buffer and stripe boundaries.

Figure 9 IOP with RAID 6 Acceleration

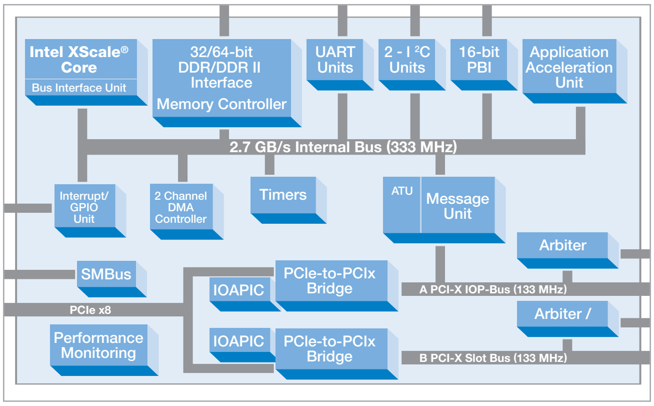


Figure 10 Data Structure of Six Source Chain Descriptor

Next Descriptor Address (NDA)
Source Address 1 (PQSAR1)
Source Address 2 (PQSAR2)
Source Address 3 (PQSAR3)
Data Multiplier Values (GFMR1)
Destination Address (DAR
Byte Count (BC)
Descriptor Control (DC)
Source Address 4 (PQSAR4)
Source Address 5 (PQSAR5)
Source Address 6 (PQSAR6)
Data Multiplier Values (GFMR2)

As mentioned in the previous sections, the GF multiplication is an important operation when computing the check value and when recovering one or two failed data drives. The AAU in the IOP333 relieves the CPU from doing time-consuming GF multiplication, thereby increasing the overall system performance. The AAU performs GF multiplications in conjunction with the XOR. The AAU multiplies source data blocks with GF field elements before the XOR operation when the P+Q RAID 6 feature is enabled. Additionally, the chain descriptor in the AAU allows the user to chain several RAID 6 computations together and offloads the RAID 6 computation to hardware instead of burdening the CPU.

The descriptor chain in the AAU provides high performance RAID 6 throughput to the RAID application. The descriptor chain is located in DDR SDRAM memory. The AAU reads the descriptor chain and performs the operations specified by it. The data structure of the P+Q chain descriptor for six source data blocks is shown in Figure 10.

NDA – contains the address of next chain descriptor

PQSAR1 – contains the source address of data block 1

PQSAR2 – contains the source address of data block 2

PQSAR3 – contains the source address of data block 3

GFMR1 – contains GF field elements for source 1 through 3 for GF multiplication

DAR – contains the destination address for XOR-ed data block

BC – contains the byte count to be XOR-ed and GF multiplied

DC – contains descriptor control bits for P+Q operations

PQSAR4 – contains the source address of data block 4

PQSAR5 – contains the source address of data block 5

PQSAR6 – contains the source address of data block 6

GMFR2 – contains GF field elements for source 4 through 6 for GF multiplication

The following sample code demonstrates the programming of a chain descriptor to setup a six source data block and P+Q check value computation using the AAU. The first routine, bld_3src_pq_desc( ) builds a 3 source descriptor entry, and the second routine, bld_6src_pq_desc( ) calls the first routine then fills in the additional 3 source data blocks in the chain descriptor.

```
 /*=================================================================
 *  Function    : bld_3src_pq_desc()
 *
 *  Arguments   : desc - contains a 3 source descriptor (empty) pointer
 *                      byte_cnt - contains the byte count of data block
 *                      parity_type - contains the type of parity to compute (P or Q)
 *                      last_entry - indicates the last entry of the descriptor.*
 *  Description : this routine sets up the P+Q xor chain descriptor for 3 sources.
 *
 *  Return      : void pointer
 * =================================================================*/
void *bld_3src_pq_desc (void *desc, unsigned int byte_cnt, parity_type parity,
                                 boolean last_entry)
{

   unsigned char *gfmulti;
   unsigned int  i;
   pq_3src  *desc_ptr;
   unsigned int  *save_nda;


   //convert void * to 3src descriptor structure
   desc_ptr = (pq_3src *)desc;

   //allocate sar1, sar2 and sar3 buffer space and fill with known data.
   alloc_buf_and_fill (byte_cnt, &(desc_ptr->sar1), (unsigned int)0x3);

   desc_ptr->gfmr1 = 0;

   //convert from word pointer to byte pointer
   gfmulti = (unsigned char *) &(desc_ptr->gfmr1);


   //use all one for coefficient if computing P parity.
   if ( parity = P_PARITY)
      desc_ptr->gfmr1 = 0x00010101;
   else
   {
      //fill in the value of GM multipler using coefficient generated
      //the lowest order byte of the word contains the data multipler for sar1,
      //the second byte for sar2 and the third byte for sar3.
      for ( i = 0; i < 3; i++)
        {
          *gfmulti = (unsigned char) gfilog [i];
             gfmulti++;
```

```
       }
   }
   //allocate memory space for destination address
   desc_ptr->dar = (unsigned int) malloc (sizeof(unsigned char) * byte_cnt);

   //set byte count in the descriptor entry
   desc_ptr->bc = byte_cnt;

   //set the descriptor control value to
   //bit 31 = 1 (dual XOR operation)                 0x80000000
   //bit 1..3 = 111 (blk#1 op=direct fill)            0x8000000E
   //bit 4..6 = 001 (blk#2 op=XOR)                   0x8000001E
   //bit 7..9 = 001 (blk#3 op=XOR)                   0x8000009E
   desc_ptr->dc |= 0x80000000;
   desc_ptr->dc |= (0x00000007 << 1);   //setting block op for blk #1
   desc_ptr->dc |= (XOR_OP << 4);      //setting block op for blk #2
   desc_ptr->dc |= (XOR_OP << 7);      //setting block op for blk #3

   save_nda = (unsigned int *)&desc_ptr->nda;
   desc_ptr++;

   if ( last_entry )
      *save_nda = (unsigned int)0;
   else
        *save_nda = (unsigned int)desc_ptr;

   return ((void *)desc_ptr);
}


/*==============================================================
 *  Function    : bld_6src_pq_desc()
 *
 *  Arguments   : desc - pointer points to 6 src descriptor (empty) entry.
 *                       byte_cnt - contains the number of bytes of data block.
 *                       parity - contains the type of parity to compute (P or Q)
 *                       last_entry - indicates this is the last entry.
 *
 *  Description : this routine sets up the P+Q xor chain descriptor for 6 sources.
 *
 *  Return      : descriptor pointer
 *
==============================================================*/
void *bld_6src_pq_desc (void *desc, unsigned int byte_cnt, parity_type parity,
                                       boolean last_entry)
{
   unsigned char *gfmulti;
   unsigned int  i;
   pq_6src       *desc_ptr;
   unsigned int  *save_nda;

   desc_ptr = (pq_6src *)desc;

   if (last_entry)
     bld_3src_pq_desc (&(desc_ptr->three_src), byte_cnt, parity, TRUE);
   else
     bld_3src_pq_desc (&(desc_ptr->three_src), byte_cnt, parity, FALSE);

   //allocate sar4, sar5 and sar6 buffer space and fill with known data.
   alloc_buf_and_fill (byte_cnt, &(desc_ptr->sar4), (unsigned int)0x3);

   desc_ptr->gfmr2 = 0;
   //convert from word pointer to byte pointer
   gfmulti = (unsigned char *) &(desc_ptr->gfmr2);

   //use all one for coefficient if computing P parity.
   if ( parity = P_PARITY)
        desc_ptr->gfmr2 = 0x00010101;
   else
   {
     //fill in the value of GM multipler using coefficient generated
     //the lowest order byte of the word contains the data multipler for sar1,
     //the second byte for src2 and the third byte for sar3.
     for ( i = 3; i < 6; i++)
       {
          *gfmulti = (unsigned char) gfilog [i];
             gfmulti++;
       }
   }

   //setting block op for blk #4
   desc_ptr->three_src.dc |= (XOR_OP << 10);
   //setting block op for blk #5
   desc_ptr->three_src.dc |= (XOR_OP << 13);
   //setting block op for blk #6
   desc_ptr->three_src.dc |= (XOR_OP << 16);
   //setting supplemental block control interpreter to 6 src.
   desc_ptr->three_src.dc |= (0x01 <<25);

   save_nda = (unsigned int *)&desc_ptr->three_src.nda;
   desc_ptr++;
   if (last_entry)
     *save_nda = (unsigned int)0;
   else
        *save_nda = (unsigned int)desc_ptr;

   return ((void *)desc_ptr);
}
```

Conclusion

A RAID 6 storage system provides enhanced data protection for critical user data. But, it is challenging to implement a RAID 6 system due to its complexity. In this paper, an overview of RAID 6 theory and implementation was presented. Additionally, the Intel IOP333 processor which accelerates RAID 6 P and Q check value computations was introduced. Users who are interested in migrating from RAID 5 to RAID 6 or designing a RAID 6 system are encouraged to use this paper as a starting point.